



下载APP



31 | 程序如何与操作系统交互？

2022-03-09 于航

《深入C语言和程序运行原理》

课程介绍 >



讲述：于航

时长 08:39 大小 7.93M



你好，我是于航。

在上一讲中我曾提到，你可以将操作系统内核暴露的“系统调用”也作为 API 的一种具体表现形式，因为调用者可以通过这些接口来使用内核提供的某种能力，但是却无需了解它们的内部实现细节。在之前的课程中，我也多次提到过有关系统调用的内容。那么，系统调用究竟是什么？它与我们编写的应用程序函数有何不同？通常情况下它又是怎样实现的呢？这一讲，我们就来看看这些问题的答案。

领资料

什么是系统调用？



不知道你还记不记得我在 [11 讲](#) “用于低级 IO 接口的操作系统调用” 小节中给出的例子，通过这个例子我们能够发现，操作系统调用实际上是由操作系统内核封装好的一些可

供上层应用程序使用的接口。这些接口为应用提供了可以按照一定规则访问计算机底层软件与硬件相关服务（如 IO、进程、摄像头等）的能力。其中，内核作为中间层，隔离了用户代码与硬件体系。

接下来，我们再通过一个简单的例子，来快速回顾下如何在 x86-64 平台上使用系统调用。在大多数情况下，位于内核之上的各类系统库（如 glibc、musl）会将这些系统调用按照不同类别进行封装，并提供可以直接在 C 代码中使用的函数接口。通过这种方式，我们就可以间接地使用系统调用。当然，在这些函数内部，系统调用的具体执行通常是由汇编指令 `syscall` 完成的。

比如，POSIX 标准中有一个名为 `getpid` 的函数，该函数用于获取当前进程的唯一标识符（ID）。如果查看 musl 中该函数针对 x86-64 平台的具体实现，你会发现这样一段代码：

 复制代码

```
1 #include <unistd.h>
2 #include "syscall.h"
3 pid_t getpid(void) {
4     return __syscall(SYS_getpid);
5 }
```

这段 C 代码作为“封装层（Wrapper）”，向上屏蔽了内部的 `sys_getpid` 系统调用。函数实现中传入的 `SYS_getpid` 是一个值为 39 的宏常量，这个值便为 `sys_getpid` 系统调用对应的唯一 ID。紧接着，通过名为 `__syscall` 的另一个函数，程序可以执行系统调用，并获取由内核返回的相关数据。这里，该函数直接由汇编代码定义，内容如下所示：

 复制代码

```
1 .global __syscall
2 .type __syscall,@function
3 __syscall:
4     movq %rdi, %rax
5     movq %rsi, %rdi
6     movq %rdx, %rsi
7     movq %rcx, %rdx
8     movq %r8, %r10
9     movq %r9, %r8
10    movq 8(%rsp), %r9
11    syscall
12    ret
```

在这段 AT&T 格式的汇编代码中，第 4 行的 `movq` 指令将传入该函数的系统调用 ID 存放到了 `rax` 寄存器中；接下来的第 5~10 行，代码将系统调用需要使用的参数也放到了相应寄存器中（这里，你可以看到 SysV 规范中对于普通函数和系统调用参数的不同处理方式）；最后，代码的第 11 行，通过 `syscall` 指令，系统调用得以被正确执行。

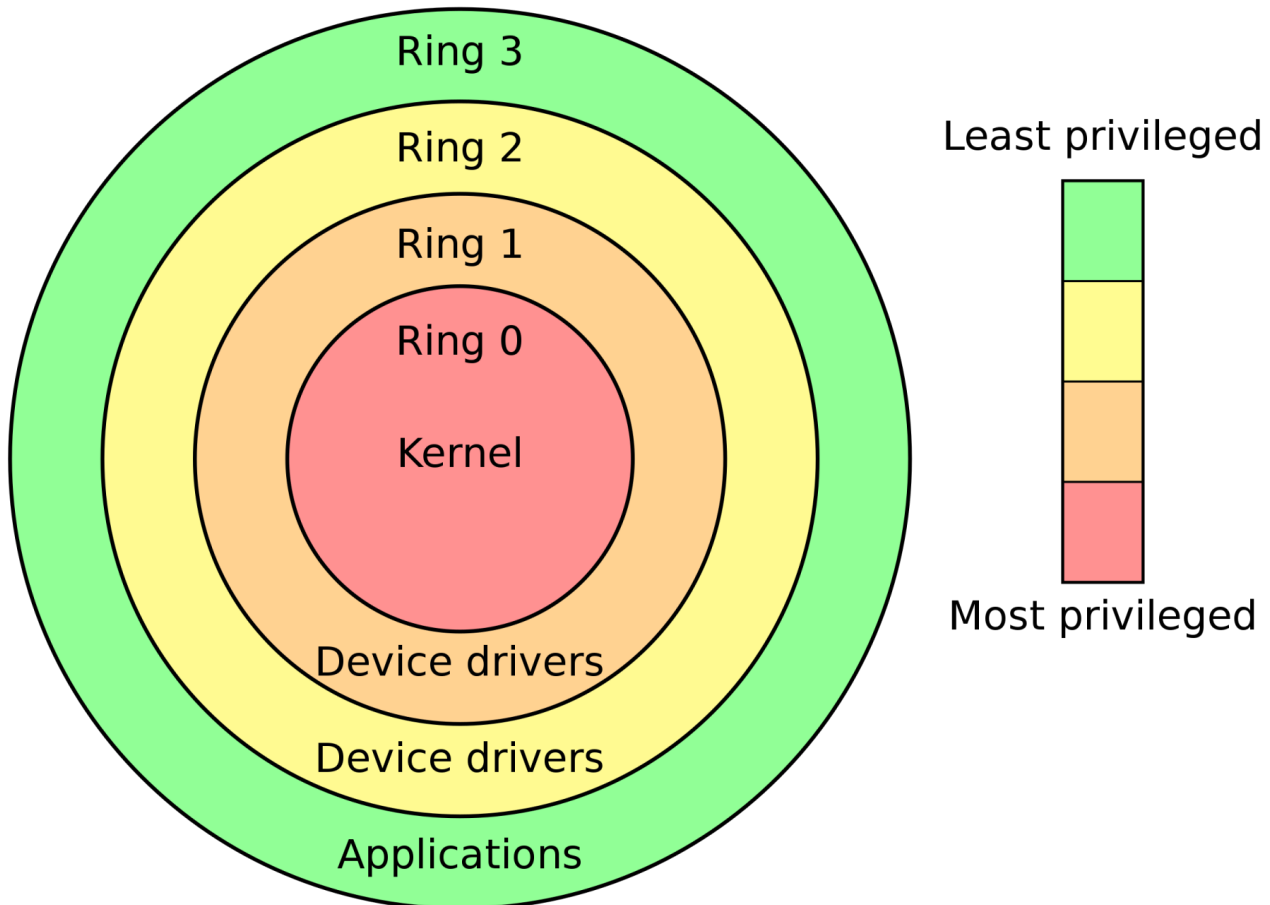
可以看到，系统调用的使用十分简单。那么，你有没有思考过这个问题：为什么系统调用需要使用特殊的 `syscall` 指令，而非“普通”的 `call` 指令进行调用呢？想要知道答案，那就要先从二者之间不同的代码执行环境开始说起了。

系统调用 vs 用户函数

系统调用与一般函数（或者说“用户函数”）的最大区别在于，**系统调用执行的代码位于操作系统底层的内核环境中，而用户函数代码则位于内核之上的应用环境中**。这两种环境有时也被称为内核态与用户态。

现代计算机通常采用名为“保护环（Protection Rings）”的机制来保护整个系统的数据和功能，使其免受故障和外部恶意行为的伤害。这种方式通过提供多种不同层次的资源访问级别，即“特权级别”，来限制不同代码的执行能力。

比如，在 Intel x86 架构中，特权级别被分为 4 个层次，即 Ring0~Ring3。其中，Ring0 层拥有最高特权，它具有对整个系统的最大控制能力，内核代码通常运行于此。相对地，Ring3 层只有最低特权，这里是应用程序代码所处的位置。而位于两者之间的 Ring1 和 Ring2 层，则通常被操作系统选择性地作为设备驱动程序的“运行等级”。你可以通过下面这张图（图片来自 [Wikipedia](#)）来直观地理解特权级别的概念。



根据特权级别的不同，CPU 能够被允许执行的机器指令以及可使用的寄存器也有所不同。比如位于 Ring3 层的应用程序，可以使用最常见的通用目的寄存器，并通过 `mov` 指令操作其中存放的数据。而位于 Ring0 层的内核代码则可以使用除此之外的 `cr0`、`cr1` 等控制寄存器，甚至通过 `in` 与 `out` 等机器指令，直接与特定端口进行 IO 操作。但如果应用程序尝试跨级别非法访问这些被限制的资源，CPU 将抛出相应异常，阻止相关代码的执行。

到这里，对于“为什么系统调用需要通过特殊的机器指令来使用”这个问题，你应该已经有了答案。系统调用是由内核提供的重要能力，而这些能力的具体实现代码属于内核代码的一部分。因此，为了执行这些代码，我们便需要一种能够在 Ring0 层将它们触发的方法，而 `syscall` 指令便能够做到这一点。

最后，再让我们来进一步看看系统调用通常是如何实现的。

系统调用的基本实现

事实上，在 x86-64 体系中，我们可以采用多种方式来执行一个系统调用。以 [@30 讲](#) 中“不遵循 ABI 的程序能否运行？”这一小节里的汇编代码为例，在不对编译命令做任何

修改的情况下，你也可以使用下面这段代码来完成同样的工作。

 复制代码

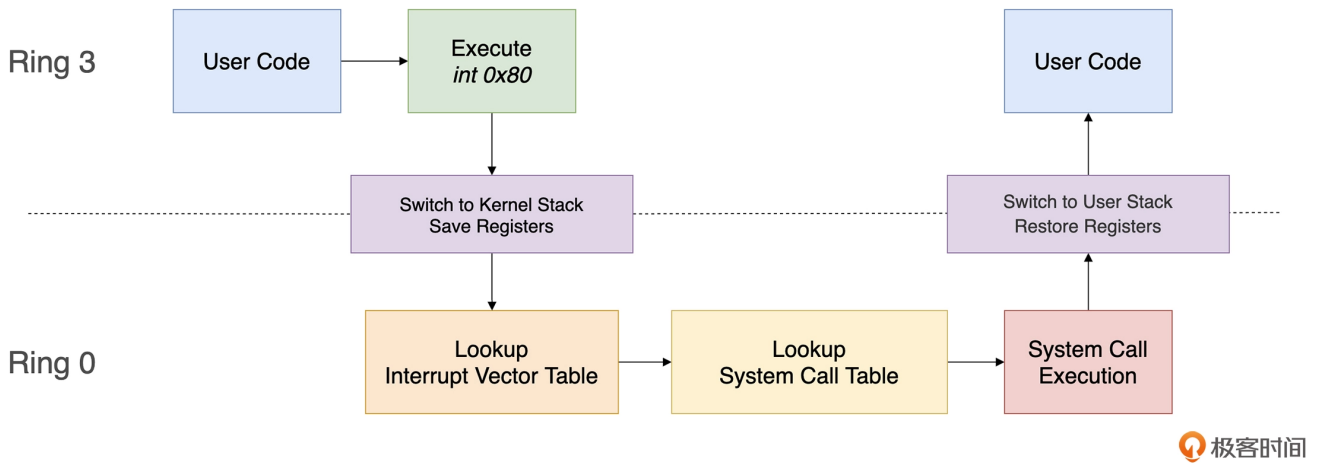
```
1 extern sub
2 global _start
3 section .text
4 _start:
5     and    rsp,0xfffffffffffffff0
6     sub    rsp, 3
7     mov    esi, 2
8     mov    edi, 1
9     call   sub
10    # use "int" to invoke a system call.
11    mov    ebx, eax
12    mov    eax, 1
13    int     0x80
```

观察上述代码的最后三行，你会发现我们触发 exit 系统调用的方式发生了变化。int 指令是一个用于产生软中断的汇编指令，它在调用时会接收一个中断号作为参数。

当中断发生时，执行环境会从 Ring3 切换至 Ring0，以准备执行内核代码。在这里，CPU 会首先根据 int 指令的参数，来从名为“中断向量表”的结构中查找下一步需要执行的中断处理程序。这里对于中断号 0x80 来说，其对应的中断处理程序便专门用于处理，由用户程序发起的系统调用请求。紧接着，这个处理程序会根据**程序通过寄存器 eax 传入的系统调用号**，来再次查找待执行的系统调用函数。最后，通过 ebx 等寄存器，系统调用函数可以获得所需参数，并完成内核某段具体代码的执行过程。

在这个过程中发生了特权级别的转换，因此，为了通过隔离执行环境来保证内核安全，CPU 在进入内核态前，通常还会进行栈的切换。比如在 Linux 中，CPU 在用户态与内核态会使用不同的栈，来分别处理发生在不同特权级别下的函数调用等过程。每一个进程都对应于独立的内核栈，这个栈中会首先存放与用户态代码执行环境相关的一系列寄存器（如 esp、eip 等）的值。而当发生在内核态的相关过程（如系统调用）结束后，进程使用的栈还需要从内核栈被再次切换回用户栈，并同时恢复保存的寄存器值。

你可以通过下图来观察上述系统调用的整个执行过程：



以上便是通过 `int` 指令来进行系统调用的大致过程。需要注意的是，虽然我们可以在 x86-64 体系上使用这种方式，但它实际仅适用于 i386 体系。在这里，我只是以这种最经典的使用方式为例，来向你展现系统调用的一种基本实现原理。

而在目前被广泛使用的 x86-64 体系中，通过 `syscall` 指令进行系统调用仍然是最高效，也最具兼容性的一种方式。`syscall` 指令的全称为“快速系统调用 (Fast System Call)”，CPU 在执行该指令时不会产生软中断，因此也无需经历栈切换和查找中断向量表等过程，执行效率会有明显的提升。

总结

这一讲，我主要带你看了三个问题，分别是什么是系统调用，系统调用与用户函数的区别，以及系统调用通常是如何实现的。

系统调用是由操作系统内核封装好的，一些可供上层应用程序使用的接口，这些接口可以让程序方便、安全地通过使用内核的能力，来间接地与底层软件和硬件进行交互。

与用户函数不同的是，系统调用需要 CPU 执行位于内核中的代码，而现代计算机采用的“保护环”机制则将整个系统的资源访问能力划分为了多个不同的特权级别。其中，Ring0 层拥有最大执行权限，它也是内核代码的运行所在。而 Ring3 层则仅有最小权限，它是上层应用程序的默认运行层级。

系统调用的经典实现方式是通过基于 `int` 指令的软中断进行的。借助软中断，CPU 可以从中断向量表中找到专门用于处理系统调用的中断处理程序，而该程序再通过由特定寄存器

传入的系统调用号，来执行相应的系统调用函数。在这个过程中，操作系统通常会进行由用户栈到内核栈的转换，以及相关寄存器的存储过程。

而在 x86-64 体系中，通过 `syscall` 指令进行的系统调用，由于不需要进行软中断和表查询，通常会有着更高的执行效率。

思考题

请查阅相关资料，了解 Linux 下的 vDSO 机制是如何参与“虚拟系统调用”的执行过程的，并在评论区说说你的理解。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

分享给需要的人，Ta 订阅超级会员，你将得 50 元

Ta 单独购买本课程，你将得 20 元

 生成海报并分享

 赞 3  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 30 | ABI 与 API 究竟有什么区别？

下一篇 春节策划一 | 构建自己的知识体系，让学习的“飞轮”持续转动

更多学习推荐



备战金三银四

Go 最新大厂面试攻略

268 道面试真题 + 大咖视频课 + 算法刷题攻略

0 元领



精选留言 (1)

写留言



Y

2022-03-09

用户空间或用户代码：运行在CPU普通权限模式 每个空间或内核代码：运行在CPU特权模式 用户空间使用内核资源：是系统调用 请问老师：如果CPU没有特权模式，是不是也就没有内核态和用户态的区别？

展开

共 1 条评论

1